Universidade federal São João Del Rei
Campus: Tancredo de Almeida Neves
Departamento de Ciência da Computação
Algoritmo e Estrutura de Dados III

Trabalho Prático:
Grafo de Fluxo de Controle

Alunos:

Luiz Felipe, 172050101
Sidney Júnior, 172050095

Professor: Alexandre Pigozzo

Agosto
2021

Conteúdo

1 Introdução

Em termos computacionais, a fase mais importante na prática de estruturas de dados é a geração do código de compilação e a otimização do mesmo. Entretanto, o estágio de compilação é responsável por gerar o código Assembly, ou código alvo, e realizar diversas otimizações durante este processo.

Para o auxílio nesta parte, usa-se uma representação nos compiladores chamada de Código de Três Endereços que é um composto por uma sequência de instruções envolvendo operações binárias e unárias,x e uma atribuição, ou seja, o nome está associado à especificação de uma instrução possui no máximo três variáveis, duas para os operadores binários e uma para o resultado.

A partir disso podemos desenvolver um Grafo de Fluxo de Controle (GFC) que é uma representação que usa a notação de grafo para descrever todos os caminhos que podem ser executados por um programa de computador, tal que, cada nó representa um bloco básico, isto é, uma região de código sequencial sem qualquer salto de execução.

2 Especificações

Observando a documentação do trabalho, pode-se obter algumas informações sobre o CTE(código de três endereços) e sobre o GFC:

2.1 Código de três endereços - CTE

É uma representação intermediária usada em back-end em que ajuda o compilador e está bem próximo a linguagem de máquina. Um endereço pode ser:

- Um nome ou identificador do programa fonte;.
- Uma constante, em que, na prática o compilador faz as conversões de tipo onde for permitido pela hierarquia de tipos;
- Um temporário que é gerado pelo compilador, na qual, é vantajoso para compiladores otimizados.

2.2 Grafo de fluxo de controle - GFC

O GFC e um grafo que contem vértices e arestas, sendo seus Vértice os blocos de código e as Arestas e as ligações possíveis entre um vértice. Com isso o GFC usa suas arestas para indicar o fluxo de controle de um no para o outro. O GFC usa Blocos básicos no qual tendem a representar uma serie de instruções de três endereços em que :

- O fluxo de controle só pode adentrar ao bloco básico por meio da primeira instrução;
- O fluxo de controle saíra do bloco somente quando alcançar na última instrução;

3 Idealização

Entre as varias possibilidades de linguagem de programação optamos por escolher umas das mais conhecida JAVA, na qual, é uma linguagem de programação orientada a objetos, em que se refere a uma área da computação chamada de Programação Orientado à Objetos (ou POO). O fator principal de escolha desta linguagem se da pelo simples fator de que um programa em JAVA pode ser armazenado em classes de forma independente, como também, podem ser carregadas no momento de utilização delas, ou seja, um programa com múltiplas linhas de execução, em que, memória que for alocada dinamicamente é desalocada de forma automática por um processo de coletor de lixo para que, evite problemas de vazamento de memória.

Dada toda complexidade do problema recorremos ao uso de **List e arrayList**, para nos ajudar na compressão do mesmo.

Com todo os pontos levantado, buscamos ajuda nos matérias e nas pesquisas e com uma ideia de como resolver o problema podemos parti para a escrita do código, testagem, correção de erros e melhoria, assim podendo chegar ao algoritmo utilizado neste trabalho.

4 O Algoritmo

O código foi desenvolvido em JAVA, pois nos permite usar o atributo de pacote, ou seja, aceita o acoplamento de classes para facilitar na organização e também o tamanho do pacote interfere na sua manutenção, onde as mudanças de software podem ser limitadas a um único pacote caso estiverem orientados a funcionalidades.

Primeiramente, para elaboração do programa foi necessário a construção da leitura do arquivo. No qual desmembramos linha a linha armazenando numa lista de instruções que logo em seguida tratamos a entrada do arquivo separando os mesmos em Atribuição, Condicional, Incondicional.

De acordo com a especificação do trabalho prático, a entrada aceita arquivo de texto que contem um Código de Três Endereços no qual foi traduzida para a linguagem de compiladores. A partir disso, varremos esse arquivo construindo o bloco básico no qual contem as instrução na qual seja líder, onde basicamente, depois dessa verificação se constrói os blocos básico que se tornam os vértices do grafo.

Com isso, para que o entendimento do algoritmo não fosse prejudicado dividimos o código em classes para cada etapa do processo como pode ser visto na Tree abaixo:

- GFC - Diretório raiz do trabalho
- bin - Diretório contendo os arquivos .class gerado pelo java
- src - Diretório principal do Trabalho
- Grafo : Diretório contendo a estrutura do Grafo.
- "Instrucao" : Diretório contendo a estrutura da nossa instrução
- Terminal : Diretório contendo a estrutura responsável pelo tratamento de entrada do arquivo a ser inserido.

4.1 SRC - Diretório principal do trabalho.

Nesse diretório se encontra a MAIN do projeto. sendo também o diretório responsável pela compilação do projeto.

4.1.1 APP.java

É a main do programa, em que, usamos seis vezes o comando import para receber as outras classes do código, para isso, criamos uma classe chamada

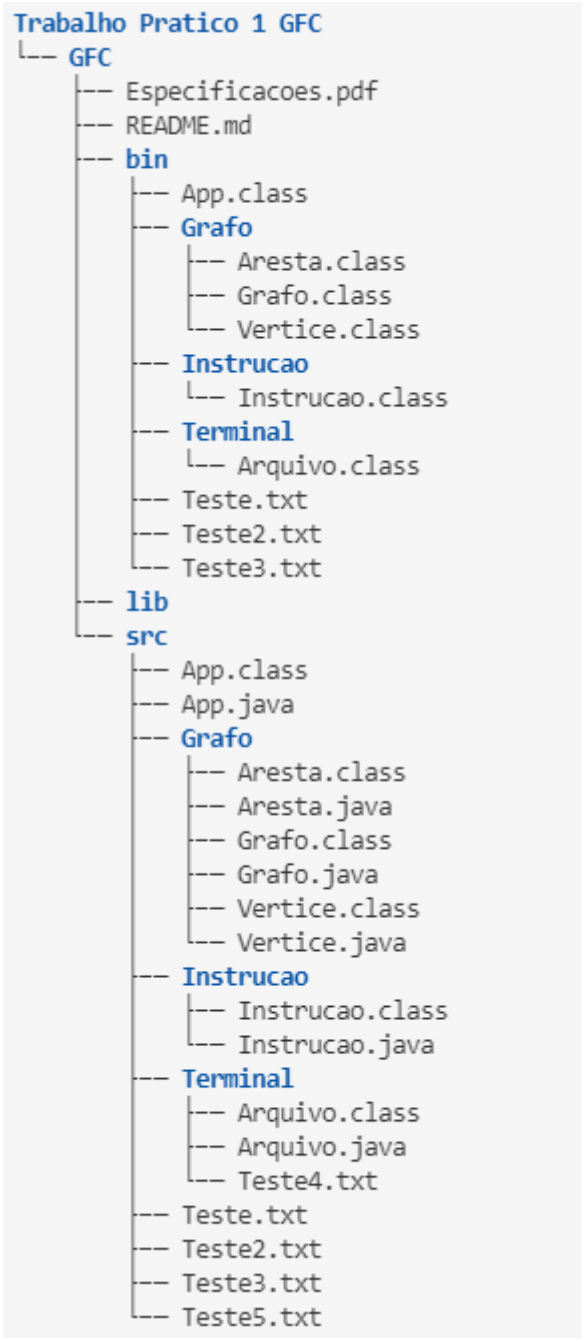

Figura 1: Tree com a estrutura do projeto

"App"onde recebe o arquivo de teste. Em seguida, criamos quatro instruções do tipo "private static"para gerar instâncias para a lista de instruções, lista de vértices, lista de arestas e para o grafo, com isso, é iniciado o ArrayList.

Em seguida, criamos uma instrução para criar o bloco de código e definir o líder das instruções, receber os vértices e arestas e seta-los no grafo, como também, imprimir o grafo criado, consequentemente, liberando a memória de instrução.

Antes de gerar o grafo, criamos uma instrução para definir as arestas do grafo, em que, criamos uma variável booleana para defini-las como verdadeiras, ou seja, criamos um "for"para cada situação. Primeiramente, precisa-se percorrer a lista de vértices, se for verdadeira, passa para o próximo "for", o segundo "for"percorre a lista de instrução pegando um bloco de código do vértice e testa se a instrução é diferente de "aspas duplas", se for ele percorre o arraylist de vértice e entra no terceiro "for". Seguindo para o terceiro "for"ele percorre o conteúdo de arraylist do vértice , em que, dentro dessa instrução percorre o arraylist das instruções, onde ele testa se o conteúdo da instrução for igual ao destino da instrução, ele adiciona uma aresta. Caso a instrução for igual a "aspas duplas"ele sinaliza como incodicional e verifica se o vértice é menor que o arraylist , caso for é adicionado uma aresta entre os vértices.

4.2 Grafo: Diretório contendo a estrutura do Grafo.

4.2.1 Grafo.java

Este pacote é responsável pela criação do Grafo do problema, em que, usamos duas bibliotecas próprias da linguagem, para o manuseio de listas de array. Criamos classes públicas para gerar a lista a partir da instrução "private static", que propaga seu valor para todas as instâncias, permitindo que retorne a variável "Vertice".

Consequentemente, para a criação das arestas usamos o mesmo esquema utilizando a instrução "private static", ou seja, para cada adição de um vértice ou de uma aresta, criamos uma função "public void" para gerar o vértice e a aresta. Após armazenar os vértices e arestas, produzimos uma função para imprimir o grafo que basicamente é gerado assim que há dois vértices e uma aresta.

4.2.2 Aresta.java

Utilizamos o pacote Grafo e esta classe é para criar as arestas do problema, em que, realizamos operações para saber a aresta inicial e a aresta final. Criamos uma função pública chamada "Aresta" para salvarmos o número para o início e para o final, em que, uma classe "Aresta" que recebe de parâmetro o número de início e o de fim, na qual, usamos a instrução "this"

que serve para referenciar o valor do objeto apontado.

Criamos duas duas funções no qual a estrutura do java pede, na qual, a "getArestaInicio" retorna o valor contido em Inicio e a "setArestaFim" retorna o valor contido em Fim. Por fim, produzimos uma função "public void" para a impressão da aresta e mostrá-la na tela.

4.2.3 Vertice.java

Utilizamos o pacote Grafo e esta classe é para criar os vértices do problema, em que, usamos três bibliotecas na qual, duas são próprias da linguagem e uma nós que implementamos. Criamos uma classe pública chamada "Vertice" com duas instruções: uma para a lista de comandos do pacote instrucoes.java e uma de identificador do vértice.

No decorrer do código, criamos instruções para receber os blocos de instruções e retornamos a instrução do bloco. Criamos uma função do tipo void "insereBloco" que recebe como parâmetro o pacote Instrução, na qual, referência a adição de instruções e uma do tipo void "setBlocos" que referencia os blocos de instrução.

Criamos uma função "Vertice" que recebe como parâmetro o inteiro de identificação, em que, atribuímos a referenciação de identificação e a atribuímos em blocos a lista de instruções. Entretanto, criamos duas funções do tipo void para a impressão das instruções e uma para imprimir o destino dessas instruções.

4.3 "Instrucao"

Diretório contendo a estrutura da nossa instrução.

4.3.1 Instrucao.java

Este pacote é responsável pelas instruções do programa onde criamos uma classe "Instrucao" na qual criamos as variáveis para receber o conteúdo das instruções, como por exemplo, do tipo string tem-se o tipo de instrução, o destino, o Label, o operando, operador1 (opFonte1) e operador2 (opFonte2) e por fim a variável de lider colocamos como booleana.

Criamos uma função para receber as variáveis que caracterizam a instrução geral do bloco, na qual, funções para cada característica da instrução e por fim, duas funções do tipo void, uma para a impressão do líder e para a impressão da instrução.

4.4 Terminal

Diretório contendo a estrutura responsável pelo tratamento de entrada do arquivo a inserido.

4.4.1 Arquivo.java

Utilizamos o pacote Terminal com as bibliotecas de .io para a leitura de arquivos e também para leitura no buffer. Criamos uma função "lendoArquivo" e criamos um bloco protegido dentro de outro, em que, primeiramente ele lê o arquivo de origem se o arquivo for diferente de NULL e o fecha em seguida retornando o texto contido no arquivo. Caso o arquivo de leitura não exista, imprime uma mensagem dizendo que houve falha após a tentativa de leitura do arquivo.

Com isso, criamos uma classe booleana estática para salvar o conteúdo do arquivo caso exista, mas caso o arquivo não existir, retorna falso.

Quando o arquivo existe lemos linha a linha e armazenamos de forma que cada linha fique num posição do array, depois retornamos e para a funcao de criaInstrucao esse array com isso nos permitindo tratar todas as condições.

4.5 Import

O comando import serve para acessar um objeto dentro da classe de um pacote a ser determinado pelo usuário, na qual, para fazer este processo precisa-se colocar o nome do pacote para importar o objeto. Os comandos usados são:

4.5.1 import java.util.List;

O termo java.util significa que será usada uma classe utilitária em que .List é basicamente que uma classe em específica é capaz de representar um objeto do tipo lista.

4.5.2 import java.util.ArrayList;

O termo java.util significa que será usada uma classe utilitária em que .ArrayList é a capacidade de agrupar objetos do tipo lista.

4.5.3 import Instrucao.Instrucao;

Esse termo se refere a capacidade de importar a classe "Instrucao"do pacote "Instrucao.java".

4.5.4 import Arquivo.Instrucao;

Esse termo se refere a capacidade de importar a classe "Instrucao"do pacote "Arquivo.java".

4.5.5 import Grafo.Aresta;

Esse termo se refere a capacidade de importar a classe "Aresta"do pacote "Grafo.java".

4.5.6 import Grafo.Grafo;

Esse termo se refere a capacidade de importar a classe "Grafo"do pacote "Grafo.java".

4.5.7 import Grafo.Vertice;

Esse termo se refere a capacidade de importar a classe "Vertice"do pacote "Grafo.java".

4.5.8 import java.io.BufferedReader;

O termo java.io se refere a classes para entrada e saída de arquivos, ou seja, java.io.BufferReader serve para melhorar a eficiência de leitura do arquivo.

4.5.9 import java.io.FileReader;

O termo java.io se refere a classes para entrada e saída de arquivos, ou seja, java.io.FileReader serve para a leitura do arquivo.

4.5.10 import java.io.IOException;

O termo java.io se refere a classes para entrada e saída de arquivos, ou seja, java.io.IOException sinaliza quando um arquivo de entrada/saída falha ou é interrompido.

5 Resultados

Os teste neste período emergencial dependia das maquinas dos membros do trabalho, sendo testados com a ajuda do WSL no Windows, na emula o sistema Ubuntu 20.04.

Os resultados obtido pelo membros, com os aquivos de teste, além de bem sucedidos,tanto usando o WSL do Windows ou com o próprio sistema Ubuntu 20.04 usado como sistema principal, fora satisfatório.

5.1 Máquina

Para os testes serem realizados foi utilizado um dos computadores pessoais dos membros do trabalho. Com as seguintes configurações:

Configurações

- **Sistema operacional:** Windows 10 - usando WSL(Bash via terminal com Ubuntu 20.04) - x64 e Ubuntu 20.04 - 64bits
- **Processador:** Intel(R) Core(TM) I7-9750H CPU 2.60GHz
- **Memoria RAM:** 8 GB

5.2 Os Testes

Para verificar a eficacia do projeto foram testados 5 arquivos, em que, cada um desses arquivos continham uma quantidade de instruções de três endereços, tendo sua saída via terminal após a execução do projeto. Recolhemos os seguintes dados dos arquivos e em seguida geramos um Grafo com a ajuda do Software JFLAP :

5.2.1 Teste 1

```
-----------------------------------------------------

BEM-VINDO AO GRAFO DE FLUXO DE CONTROLE:
O ARQUIVO DE TESTE E: Teste.txt

-----------------------------------------------------
 IMPRIMINDO OS BLOCO COM OS CODIGO DE TRES ENDERECOS
-----------------------------------------------------

B1:
x=0
y=1
t0=32
t1=t0x
x=t1
t2=xy
 t2!=0gotoL1

B2:
 y==0gotoL2

B3:
L1: z=1
gotoL0

B4:
L2: z=0

B5:
L0

-----------------------------------------------------
IMPRIMINDO O GRAFO DE FLUXO DE CONTROLE:
-----------------------------------------------------

Entry ARESTA VAI PARA (-->) B1
B1 ARESTA VAI PARA (-->) B3
B1 ARESTA VAI PARA (-->) B2
B2 ARESTA VAI PARA (-->) B4
```

```
B2 ARESTA VAI PARA (-->) B3
B3 ARESTA VAI PARA (-->) B5
B4 ARESTA VAI PARA (-->) B5
B5 ARESTA VAI PARA (-->) EXIT

-----------------------------------------------------
```

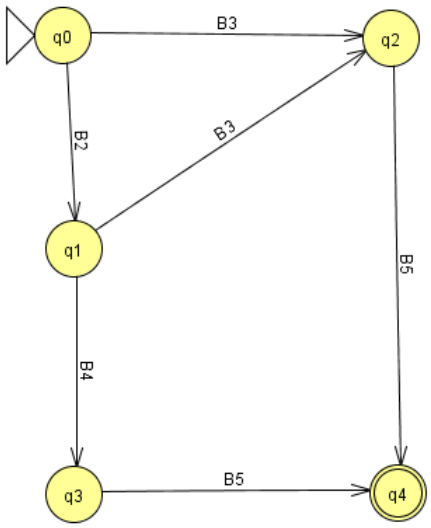

Figura 2: Grafo do teste 1 montado com ajuda do JFLAP

Depois de gerado o GFC temos 2 perguntas para responder sendo elas

- 1 - A partir do GFC construído, quais são os possíveis fluxos de execução do programa?

  No caso do teste 1 podemos identificar o fluxo de execução sendo

- B1 -≫ B3 -≫ B5
- B1 -≫ B2 -≫ B3 -≫ B5
- B1 -≫ B2 -≫ B4 -≫ B5
- 2 - Quais são os loops no GFC caso existam?

  No caso do teste 1 não conseguimos identificar ou pode não haver loop.

5.2.2 Teste 2

```
-----------------------------------------------------

BEM-VINDO AO GRAFO DE FLUXO DE CONTROLE:
O ARQUIVO DE TESTE E: Teste2.txt

-----------------------------------------------------
 IMPRIMINDO OS BLOCO COM OS CODIGO DE TRES ENDERECOS
-----------------------------------------------------

B1:
 2!=0gotoL1

B2:
 4==0gotoL2:

B3:
L1: z=1
gotoL0

B4:
L2: z=0

B5:
L0

-----------------------------------------------------
IMPRIMINDO O GRAFO DE FLUXO DE CONTROLE:
-----------------------------------------------------

Entry ARESTA VAI PARA (-->) B1
B1 ARESTA VAI PARA (-->) B3
B1 ARESTA VAI PARA (-->) B2
B2 ARESTA VAI PARA (-->) B3
B3 ARESTA VAI PARA (-->) B5
B4 ARESTA VAI PARA (-->) B5
B5 ARESTA VAI PARA (-->) EXIT

-----------------------------------------------------
```

Depois de gerado o GFC temos 2 perguntas para responder sendo elas

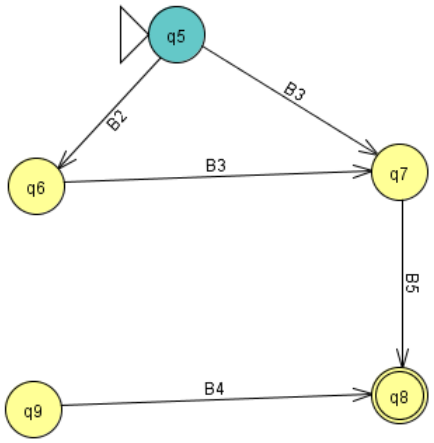

Figura 3: Grafo do teste 2 montado com ajuda do JFLAP

- 1 - A partir do GFC construído, quais são os possíveis fluxos de execução do programa?

  No caso do teste 2 podemos identificar o fluxo de execução sendo

- B1 -≫ B3 -≫ B5
- B1 -≫ B2 -≫ B3 -≫ B5
- B4 -≫ B5
- 2 - Quais são os loops no GFC caso existam?

  No caso do teste 2 não conseguimos identificar ou pode não haver loop.

5.2.3 Teste 3

```
-----------------------------------------------------

BEM-VINDO AO GRAFO DE FLUXO DE CONTROLE:
O ARQUIVO DE TESTE E: Teste3.txt

-----------------------------------------------------
 IMPRIMINDO OS BLOCO COM OS CODIGO DE TRES ENDERECOS
-----------------------------------------------------

B1:
s=0
i=0
k=2

B2:
L0:  i<10gotoL2

B3:
j=0

B4:
L3:  j<10gotoL1

B5:
t0=sk
s=t0
t1=j1
j=t1
gotoL3

B6:
L1: t2=i1
i=t2
gotoL0

B7:
L2

-----------------------------------------------------
```

```
IMPRIMINDO O GRAFO DE FLUXO DE CONTROLE:
-------------------------------------------------------

Entry ARESTA VAI PARA (-->) B1
B1 ARESTA VAI PARA (-->) B2
B2 ARESTA VAI PARA (-->) B7
B2 ARESTA VAI PARA (-->) B3
B3 ARESTA VAI PARA (-->) B4
B4 ARESTA VAI PARA (-->) B6
B4 ARESTA VAI PARA (-->) B5
B5 ARESTA VAI PARA (-->) B4
B6 ARESTA VAI PARA (-->) B2
B7 ARESTA VAI PARA (-->) EXIT

-------------------------------------------------------
```

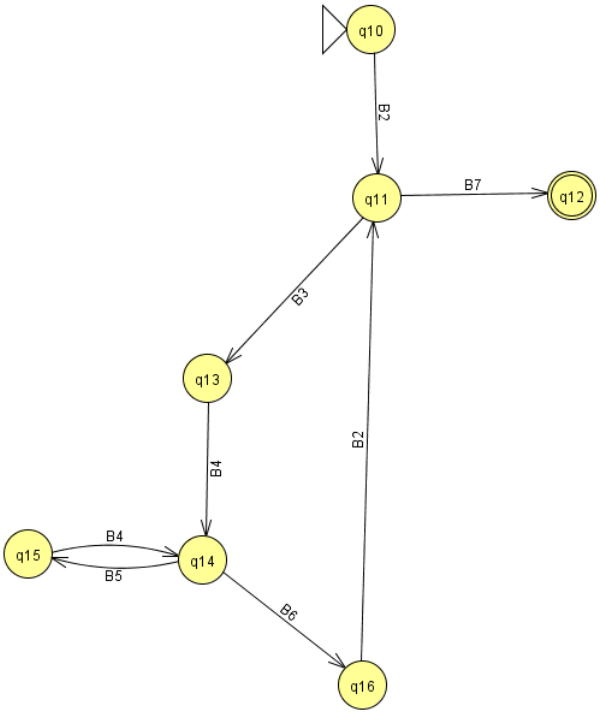

Figura 4: Grafo do teste 3 montado com ajuda do JFLAP

Depois de gerado o GFC temos 2 perguntas para responder sendo elas

- 1 - A partir do GFC construído, quais são os possíveis fluxos de execução do programa?

  No caso do teste 3 podemos identificar o fluxo de execução sendo

- B1 -≫ B2 -≫ B7
- B1 -≫ B2 -≫ B3 -≫ B4 -≫ B5 -≫ B4-≫ B6 -≫ B2 -≫ B7
- B1 -≫ B2 -≫ B3 -≫ B4 -≫ B6 -≫ B2 -≫ B7
- 2 - Quais são os loops no GFC caso existam?

  no caso do teste 3 podemos identificar 2 loop sendo eles

- B4 -≫ B5 -≫ B4
- B2 -≫ B3 -≫ B4 -≫ B6 -≫ B2

5.2.4 Teste 4

```
-----------------------------------------------------

BEM-VINDO AO GRAFO DE FLUXO DE CONTROLE:
O ARQUIVO DE TESTE E: Teste4.txt

-----------------------------------------------------
 IMPRIMINDO OS BLOCO COM OS CODIGO DE TRES ENDERECOS
-----------------------------------------------------

B1:
x=1
y=10
a=2
z=1000
 y<1gotoL2

B2:
L1: t0=y1
y=t0
t0=x2
x=t0
t0=z/x
z=t0
 y<1gotoL1

B3:
L2

-----------------------------------------------------
IMPRIMINDO O GRAFO DE FLUXO DE CONTROLE:
-----------------------------------------------------

Entry ARESTA VAI PARA (-->) B1
B1 ARESTA VAI PARA (-->) B3
B1 ARESTA VAI PARA (-->) B2
B2 ARESTA VAI PARA (-->) B2
B2 ARESTA VAI PARA (-->) B3
B3 ARESTA VAI PARA (-->) EXIT
```

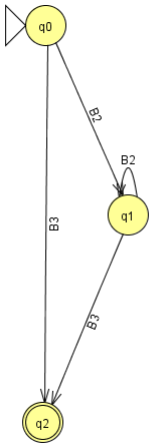

Figura 5: Grafo do teste 4 montado com ajuda do JFLAP

Depois de gerado o GFC temos 2 perguntas para responder sendo elas

- 1 - A partir do GFC construído, quais são os possíveis fluxos de execução do programa?

  No caso do teste 4 podemos identificar o fluxo de execução sendo

- B1 -≫ B2 -≫ B3
- B1 -≫ B2 -≫ B2 -≫ B3
- B1 -≫ B3
- 2 - Quais são os loops no GFC caso existam?

  No caso do teste 4 podemos identificar 1 loop ou self loop sendo ele

- B2 -≫ B2

5.2.5 Teste 5

```
-----------------------------------------------------

BEM-VINDO AO GRAFO DE FLUXO DE CONTROLE:
O ARQUIVO DE TESTE E: Teste5.txt

-----------------------------------------------------
 IMPRIMINDO OS BLOCO COM OS CODIGO DE TRES ENDERECOS
-----------------------------------------------------

B1:
c=0
gotoL2

B2:
L1: c=1

B3:
L2

-----------------------------------------------------
IMPRIMINDO O GRAFO DE FLUXO DE CONTROLE:
-----------------------------------------------------

Entry ARESTA VAI PARA (-->) B1
B1 ARESTA VAI PARA (-->) B3
B2 ARESTA VAI PARA (-->) B3
B3 ARESTA VAI PARA (-->) EXIT

-----------------------------------------------------
```

Depois de gerado o GFC temos 2 perguntas para responder sendo elas

- 1 - A partir do GFC construído, quais são os possíveis fluxos de execução do programa?

  No caso do teste 5 podemos identificar o fluxo de execução sendo:

- B1 -≫ B3
- B4 -≫ B3

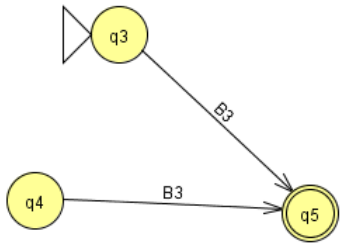

Figura 6: Grafo do teste 5 montado com ajuda do JFLAP

- 2 - Quais são os loops no GFC caso existam?

  No caso do teste 5 não podemos identificar nenhum loop ou não existe.

6 Conclusão

Analisando a complexidade inicial do problema proposto para este trabalho, os testes utilizando a linguagem C desde a abertura do arquivo até a impressão do mesmo, se mostrariam muito ineficientes,pois essa técnica é simples de ser implementada porem é ineficiente em quesito de custo computacional, inviabilizando seu uso em problemas desse tipo.

O uso da linguagem JAVA para o desenvolvimento do código trás vantagens, uma delas é a possibilidade do uso de instancias e funções do tipo públicas e privadas, na qual, torna o mais simples de implementar, como também, a existência das funções que facilitam as comparações.

Visto a complexidade do problema, o uso de uma linguagem mais robusta torna-se um facilitador para se idealizar um modelo de partida para a criação do algoritmo que o representa. Logo foi de extrema importância para facilitar a visualização do problema que de inicio se mostrava muito complexo.

Conclui-se que com o uso de uma linguagem própria da área de programação orientada a objetos para desenvolver este trabalho, pois notamos que o código ficaria mais legível e entendível.

Referências

1) N. ZIVIANI, Projeto de Algoritmos com Implementações em Pascal e C, 3a edição Editora Cengage Learning, 2010

2) C.N. Lintzmayer, Mota, G.O. Análise de algoritmos e estruturas de dados, 2021+

3) ZIVIANI, Nivio. Projeto de algoritmos: com implementações em Java e C++. São Paulo: Editora Cengage Learning, 2007.

4)T. H. CORMEN, C. E. LEISERSON, R. L. RIVEST, C. STEIN, Algoritmos, Teoria e Prática, Campus, 2002.

5) UFPE. Pacotes e Bibliotecas de Java: Programação 3: Orientação a Objetos e Java. Cin UFPE. Pernambuco.
Disponível em: https://www.cin.ufpe.br/ if101/turmaatual/aulas/aula8/transparencias/pacotes.l
Acesso em: 9 jul. 2021..

6) INTRODUÇÃO à Linguagem Java Através de Exemplos. UFCG. Paraíba.
Disponível em: http://www.dsc.ufcg.edu.br/ jacques/cursos/p2/html/intro/intro.htm.
Acesso em: 12 jul. 2021.

7) BALLEM. Fundamentos da Instrução Package. MBALLEM — PROGRAMANDO COM JAVA BLOG.
Disponível em: https://www.mballem.com/post/fundamentos-da-instrucao-package/. Acesso em: 14 jul. 2021.

8) BALLEM. Fundamentos da Instrução Import. MBALLEM — PROGRAMANDO COM JAVA BLOG.
Disponível em: https://www.mballem.com/post/fundamentos-da-instrucao-import/. Acesso em: 15 jul. 2021.

9) JUNIOR, Vilson Heck. Java: Procedimentos, Funções e Métodos. IFSC.
Disponível em: http://docente.ifsc.edu.br/vilson.junior/ed/Rev$_{J}ava_{M}etodos.pdf.Acessoem$: $24jul.2021$.